Poll: Who is responsible for erasing a whiteboard in a public space?

a) The person who is done using it
b) The person who is about to use it

EECS 370 - Lecture 6

Function Calls

Announcements

- P1
  - Part a due tonight @ 11:55 via Autograder
  - Project 1 s + m due next Thu

- HW 1
  - Due next-next Monday
- Lab 3 meets Fr/M
- Get exam conflicts and SSD accommodations sent to us **in the next week**
  - Forms listed on the website

Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: ISA - storage types, binary and addressing modes
- Lecture 3 : LC2K
- Lecture 4 : ARM
- Lecture 5 : Converting C to assembly – basic blocks
- **Lecture 6 : Converting C to assembly – functions**
- Lecture 7 : Translation software; libraries, memory layout

Agenda

- **Using branches more generally**
- Function calls and the call stack
- Assigning variables to memory locations
- Saving registers
- Caller/callee example

Branching far away

- Underlying philosophy of ISA design: **make the common case fast**
- Most branches target nearby instructions
  - Displacement of 19 bits is usually enough
- BUT what if we need to branch really far away (more than 2^{19} words)?

```
        CBZ     X15, FarLabel
```

- The assembler is smart enough to replace that with

```
        CBNZ    X15, L1
        B       FarLabel
L1:     ........
```

replace

- The simple branch instruction (B) has a 26 bit offset which spans about 64 million instructions!
- In LC2K, we can do a similar thing by using JALR instead of BEQ

Unconditional Branching Instructions

Unconditional branch	branch	B 2500	go to PC + 10000	Branch to target address; PC-relative
	branch to register	BR X30	go to X30	For switch, procedure return
	branch with link	BL 2500	X30 = PC + 4; PC + 10000	For procedure call PC-relative

we don't specify X30 here → So X30 is implicity been used.

number of instruction

each instruction is 4 bytes

this hold the memory address

- There are three types of unconditional branches in the LEGv8 ISA.
  - The first **(B)** is the PC relative branch with the 26 bit offset from the last slide.
  - The second **(BR)** jumps to the address contained in a register (X30 above)
  - The third **(BL)** is like our PC relative branch but it does something else.
    - It sets X30 (always) to be the current PC+4 before it branches.

Look inside the register which hold the address that we want to branch to. (64 bits)

- Why is BL storing PC+4 into a register?

Branch with Link (BL)

- Branch with Link is the branch instruction used to call functions
  - Functions need to know where they were called from so they can return.
    - In particular they will need to return to right after the function call
    - Can use "BR X30" saving where you are, when you return from the function, you know where you go.

means 200 instruction

- Say that we execute the instruction BL # $\boxed{200}$ when at PC 1000.
  - What address will be branched to? 1800
  - What value is stored in X30? $X_{30} = 1004$
  - How is that value in X30 useful?

we will finally branch back to X30

1000 : BL # 200 $\xrightarrow{\text{go to}}$ $1000 + 4 \times 200 = 1800$

Poll

Agenda

- Using branches more generally
- **Function calls and the call stack**
- Assigning variables to memory locations
- Saving registers
- Caller/callee example

Converting function calls to assembly code

C: factorial(5);

- Need to pass parameters to the called function—factorial
- Need to save return address of caller so we can get back
- Need to save register values (why?)
- Need to jump to factorial

Execute instructions for factorial()
Jump to return address

- Need to get return value (if used)
- Restore register values

Task 1: Passing parameters

- Where should you put all of the parameters?
  - Registers?
    - Fast access but few in number and wrong size for some objects
  - Memory?
    - Good general solution but slow
- ARMv8 solution—and the usual answer:
  - Both
    - Put the first few parameters in registers (if they fit) (X0 – X7) 8
    - Put the rest in memory on the call stack— important concept

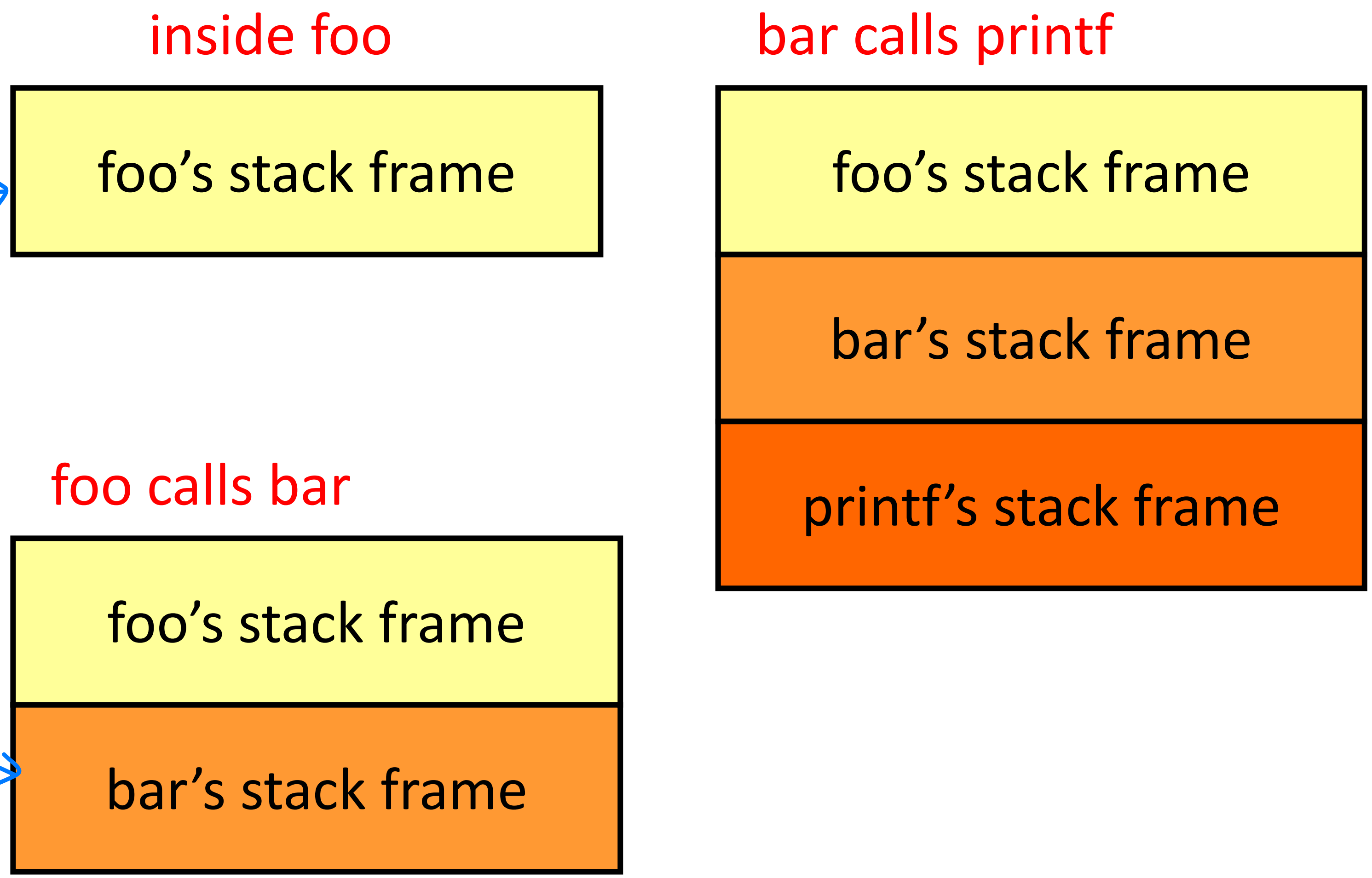
Call stack

- ARM conventions (and most other processors) allocate a region of memory for the "call" stack
  - This memory is used to manage all the storage requirements to simulate function call semantics
    - Parameters (that were not passed through registers)
    - Local variables
    - Temporary storage (when you run out of registers and need somewhere to save a value)
    - Return address ← Before we call other function, we need to store the reture address for this level function.
    - Etc.
- Sections of memory on the call stack [**stack frames**] are allocated when you make a function call, and de-allocated when you return from a function

The stack grows as functions are called

FUNCTION CALLS

```c
void foo()
{
   int x, y[2];
   bar(x);
}

void bar(int x)
{
   int a[3];
   printf();
}
```



The stack shrinks as functions return

FUNCTION CALLS

```
void foo()
{
  int x, y[2];
  bar(x);
}

void bar(int x)
{
  int a[3];
  printf();
}
```

printf returns

foo's stack frame
bar's stack frame

bar returns

foo's stack frame

Stack frame contents

FUNCTION CALLS

```c
void foo()
{
  int x, y[2];
  bar(x);
}

void bar(int x)
{
  int a[3];
  printf();
}
```

foo was called by main

foo's stack frame

return addr to main
x
y[0]
y[1]
spilled registers in foo

as we doing internal computation if they don't fit inside of our own registers .

Stack frame contents (2)

FUNCTION CALLS

foo calls bar

```
void foo()
{
  int x, y[2];
  bar(x);
}

void bar(int x)
{
  int a[3];
  printf();
}
```

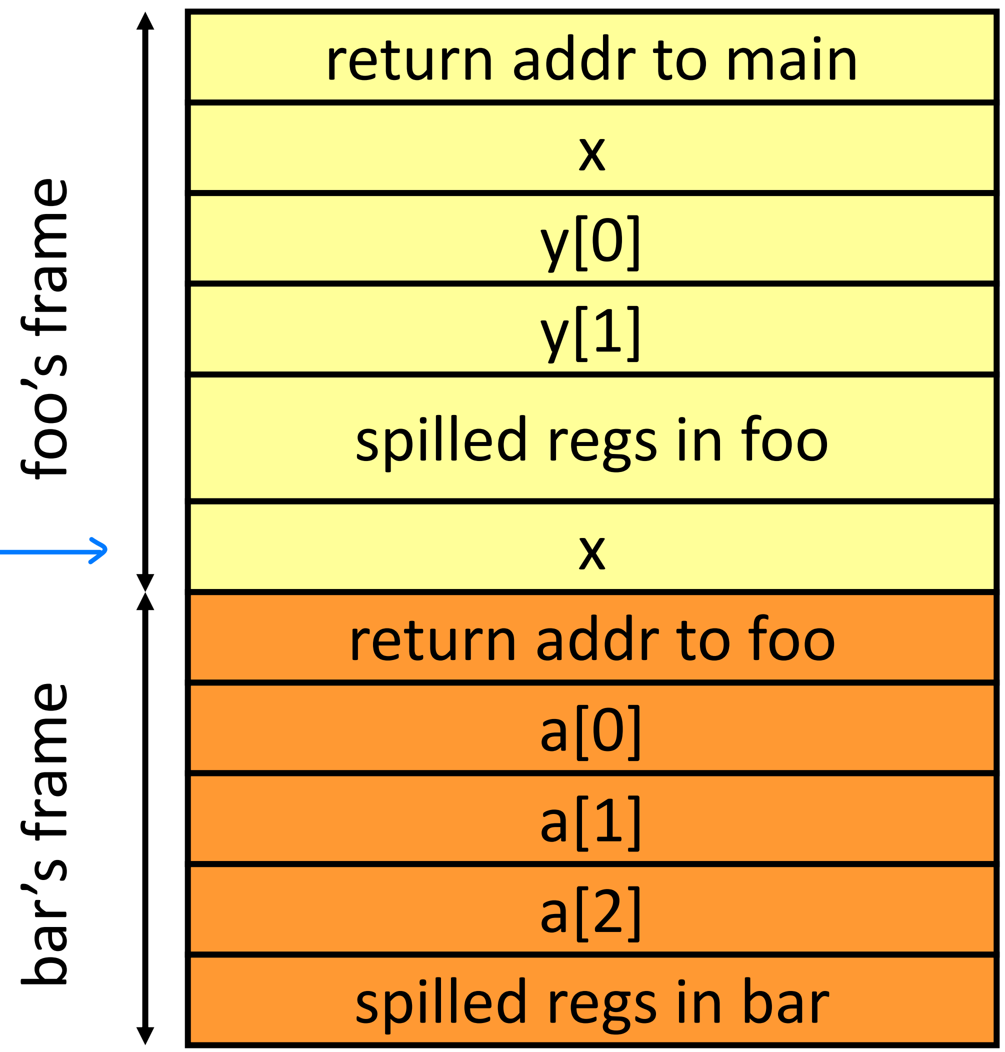
foo would write x to this location to pass the parameter to Bar,

And Bar can read from that location to see what that parameter is.

two copy of x:

① foo has it's local copy

② bar also has it.



Spill data-—not enough room in x0-x7 for params and also caller and callee saves

Agenda

- Using branches more generally
- Function calls and the call stack
- **Assigning variables to memory locations**
- Saving registers
- Caller/callee example

Review: Where do the variables go?

Assigning variables to memory spaces

FUNCTION CALLS

global variable

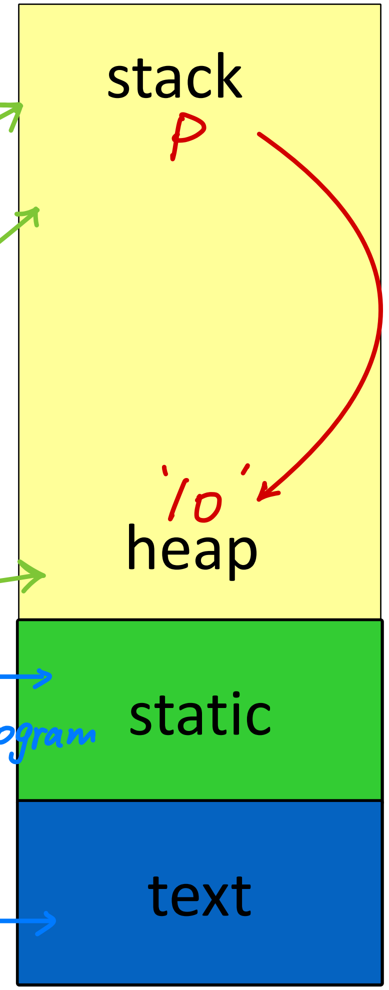
```
int w;
void foo(int x)
{
  static int y[4];
  char* p;
  p = malloc(10);
  //…
  printf("%s\n", p);
}
```

one istance of this variable that is shared cross all function calls.

p is not dynamic memory

p is a pointer that will hold the address of something in dynamic memory.

But the pointer itself is just a normal declared variable.



Any object whose lifetime is whole program

hold all of the machine code, instructions. (programe)

Assigning variables to memory spaces

FUNCTION CALLS

```
int w;
void foo(int x)
{
  static int y[4];
  char* p;
  p = malloc(10);
  //…
  printf(“%s\n”, p);
}
```

w goes in static, as it’s a global

x goes on the stack, as it’s a parameter

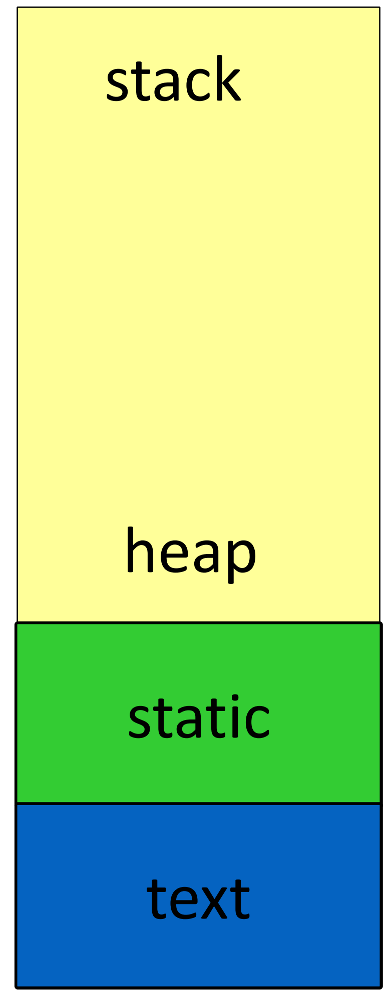
y goes in static, 1 copy of this!!

p goes on the stack

allocate 10 bytes on heap, ptr set to the address

string goes in static, pointer to string on stack, p goes on stack

bigger



smaller

The addresses of local variables will be different depending on where we are in the call stack

So we can't hard code the address

Accessing Local Variables

- Stack pointer (SP):
  - register that keeps track of current top of stack
- Compiler (or assembly writer) knows relative offsets of objects in stack
- Can access using lw/sw offsets

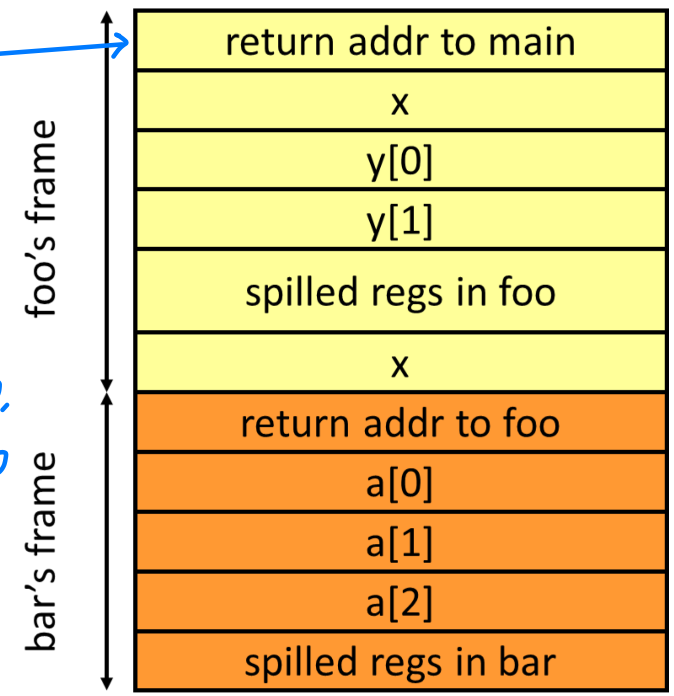
spacific register

```
sw   x30   [sp, #0]
sw   x1    [sp, #4]
sw   x2    [sp, #8]
```

All the address is related to sp when you express them

sp

when we go to foo, the sp will point to top of foo



Agenda

- Using branches more generally
- Function calls and the call stack
- Assigning variables to memory locations
- **Saving registers**
- Caller/callee example

What about registers?

- Higher level languages (like C/C++) provide many abstractions that don't exist at the assembly level
- E.g. in C, each function has its own local variables
  - Even if different function have local variables with the same name, they are independent and guaranteed not to interfere with each other!

```
void foo(){
  int a=1;
  bar();
  printf(a);
}
```

Still prints "1"… these don't interfere

```
void bar(){
  int a=2;
  return;
}
```

What about registers?

- But in assembly, all functions share a small set (e.g. 32) of registers
  - Called functions will overwrite registers needed by calling functions

foo() overwrites X0 if we don't do something!!

```
main: movz X0, #1
      bl foo
      bl printf
```

But in foo, may let $X_0 = 2$

```
foo: movz X0, #2
     br X30
```

- "Someone" needs to save/restore values when a function is called to ensure this doesn't happen

Two Possible Solutions

- Either the **called** function saves register values before it overwrites them and restores them before the function returns (**callee** saved)…

```
main: movz X0, #1
      bl foo
      bl printf
```

Store X_0 out to Stack

```
foo: stur X0, [stack]
     movz X0, #2
     ldur X0, [stack]
     br X30
```

load back to X_0

- Or the **calling** function saves register values before the function call and restores them after the function call (**caller** saved)…

```
main: movz X0, #1
      stur X0, [stack]
      bl foo
      ldur X0, [stack]
      bl printf
```

```
foo: movz X0, #2
     br X30
```

a->r1, b->r2, etc.

Another example

Original C Code

```
void foo(){
  int a,b,c,d;

  a = 5; b = 6;
  c = a+1; d=c-1;

  bar();

  d = a+d;
  return();
}
```

only reading two of them after bar has returned

No need to save r2/r3. Why?

Additions for Caller-save

```
void foo(){
  int a,b,c,d;
  r1
  a = 5; b = 6;
  c = a+1; d=c-1;   r4
  save r1 to stack
  save r4 to stack
  bar();
  restore r1
  restore r4
  d = a+d;
  return();
}
```

Assume bar() will overwrite registers holding a,d

Additions for Callee-save

```
void foo(){
  int a,b,c,d;
  save r1
  save r2
  save r3
  save r4
  a = 5; b = 6;
  c = a+1; d=c-1;
  bar();
  d = a+d;
  restore r1
  restore r2
  restore r3
  restore r4
  return();
}
```

bar() will save a,b, but now foo() must save main's variables

“caller-save” vs. “callee-save”

- Caller-save
  - What if bar() doesn't use r1/r4?
  - No harm done, but wasted work

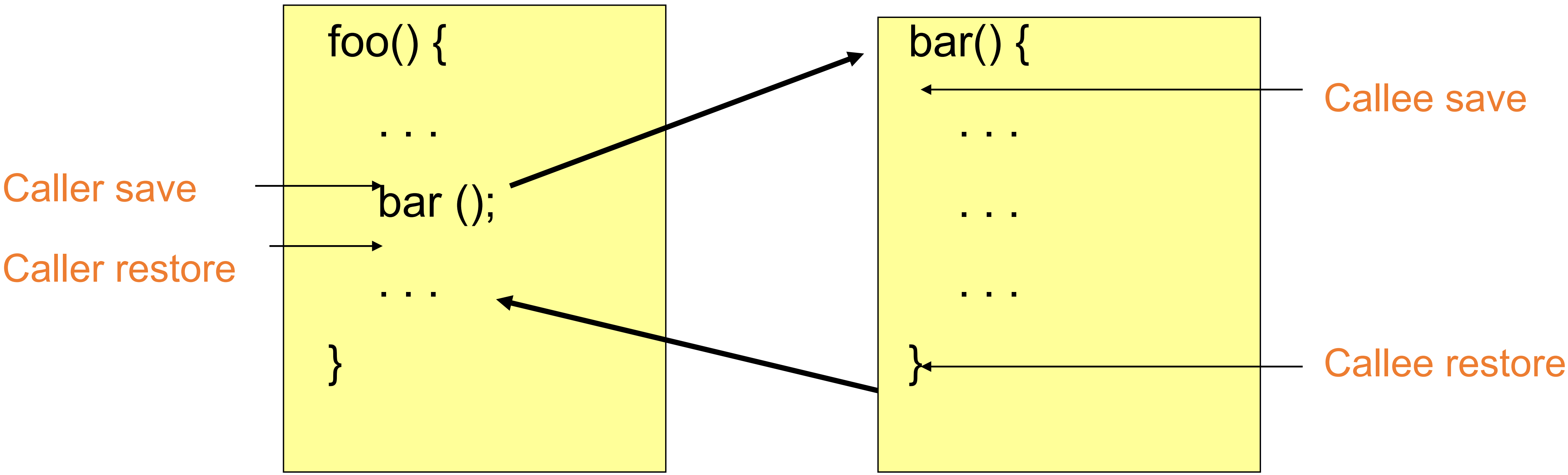
```
void foo(){
  int a,b,c,d;

  a = 5; b = 6;
  c = a+1; d=c-1;
  save r1 to stack
  save r4 to stack
  bar();
  restore r1
  restore r4
  d = a+d;
  return();
}
```

- Callee-save
  - What if main() doesn't use r1-r4?
  - No harm done, but wasted work

```
void foo(){
  int a,b,c,d;
  save r1
  save r2
  save r3
  save r4
  a = 5; b = 6;
  c = a+1; d=c-1;
  bar();
  d = a+d;
  restore r1
  restore r2
  restore r3
  restore r4
  return();
}
```

Another helpful visual

CALLER-CALLEE



Saving/Restoring Optimizations

CALLER-CALLEE

- Where can we avoid loads/stores?
- Caller-saved
  - Only needs saving if value is "live" across function call
  - Live = contains a useful value: Assign value before function call, use that value after the function call
  - In a leaf function (a function that calls no other function), caller saves can be used without saving/restoring

a, d are live

b, c are NOT live

```
void foo(){
  int a,b,c,d;

  a = 5; b = 6;
  c = a+1; d=c-1;

  bar();

  d = a+d;
  return();
}
```

Saving/Restoring Optimizations

CALLER-CALLEE

- Where can we avoid loads/stores?
- Callee-saved
  - Only needs saving at beginning of function and restoring at end of function
  - Only save/restore it if function overwrites the register

Only use r1-r4

No need to save other registers

```
void foo(){
  int a,b,c,d;

  a = 5; b = 6;
  c = a+1; d=c-1;

  bar();

  d = a+d;
  return();
}
```

Agenda

- Using branches more generally
- Function calls and the call stack
- Assigning variables to memory locations
- Saving registers
- **Caller/callee example**

Caller versus Callee

- Which is better??
- Let's look at some examples...

- Simplifying assumptions:
  - A function can be invoked by many different call sites in different functions.
  - Assume no inter-procedural analysis (hard problem)
    - A function has no knowledge about which registers are used in either its caller or callee
    - Assume main() is not invoked by another function
  - Implication
    - Any register allocation optimization is done using function local information

Caller-saved vs. callee saved – Multiple function case

```
void main(){
  int a,b,c,d;

  c = 5; d = 6;
  a = 2; b = 3;
  foo();
  d = a+b+c+d;
}
```

```
void foo(){
  int e,f;

  e = 2; f = 3;
  bar();
  e = e + f;
}
```

```
void bar(){
  int g,h,i,j;

  g = 0; h = 1;
  i = 2; j = 3;
  final();
  j = g+h+i;
}
```

```
void final(){
  int y,z;

  y = 2; z = 3;

  z = y+z;
}
```

Note: assume main does not have to save any callee registers

Caller-saved vs. callee saved – Multiple function case

- Questions:

1. How many registers need to be saved/restored if we use a **caller-save** convention?
2. How many registers need to be saved/restored if we use a **callee-save** convention?
3. How many registers need to be saved/restored if we use a mix of **caller-save** and **callee-save**?

Poll: How many ld/st pairs are needed?

Question 1: Caller-save

```
void main(){
  int a,b,c,d;
  c = 5; d = 6;
  a = 2; b = 3;
  [4 STUR]
  foo();
  [4 LDUR]
  d = a+b+c+d;
          4
}
```

```
void foo(){
  int e,f;

  e = 2; f = 3;
  [2 STUR]
  bar();
  [2 LDUR]
  e = e + f;
          2
}
```

```
void bar(){
  int g,h,i,j;
  g = 0; h = 1;
  i = 2; j = 3;
  [3 STUR]
  final();
  [3 LDUR]
  j = g+h+i;
        3
}
```

```
void final(){
  int y,z;


  y = 2; z = 3;

  z = y+z;

}
```

Total: 9 STUR / 9 LDUR

Question 2: Callee-save

Poll: How many ld/st pairs are needed?

```
void main(){
  int a,b,c,d;

  c = 5; d = 6;
  a = 2; b = 3;
  foo();
  d = a+b+c+d;



}
```

```
void foo(){
  [2 STUR]
  int e,f;

  e = 2; f = 3;
  bar();          2
  e = e + f;



  [2 LDUR]
}
```

```
void bar(){
  [4 STUR]
  int g,h,i,j;
  g = 0; h = 1;
  i = 2; j = 3; 4
  final();
  j = g+h+i;



  [4 LDUR]
}
```

```
void final(){
  [2 STUR]
  int y,z;

  y = 2; z = 3;
                2
  z = y+z;



  [2 LDUR]
}
```

Total: 8 STUR / 8 LDUR

Is one better?

- Caller-save works best when we don't have many live values across function call
- Callee-save works best when we don't use many registers overall
- We probably see functions of both kinds across an entire program
- Solution:
  - Use both!
  - E.g. if we have 6 registers, use some (say r0-r2) as caller-save and others (say r3-r5) as callee-save
  - Now each function can optimize for each situation to reduce saving/restoring

Poll: What's the optimal number of caller/callee registers for final?

Question 3: Mixed 3 caller / 3 callee

- For main, ideally put all variables in callee-save registers
- But we only 3 are available
- One variable needs to go in caller-saved register

```
void main(){
  int a,b,c,d;
  c = 5; d = 6;
  a = 2; b = 3;
  [1 STUR]
  foo();
  [1 LDUR]
  d = a+b+c+d;
}
```

1 caller r
3 callee r

```
void foo(){
  [2 STUR]
  int e,f;

  e = 2; f = 3;
  bar();
  e = e + f;

  [2 LDUR]
}
```

2 callee r
(2 caller would be equivalent)

```
void bar(){
  [3 STUR]
  int g,h,i,j;
  g = 0; h = 1;
  i = 2; j = 3;
  final();
  j = g+h+i;

  [3 LDUR]
}
```

1 caller r
3 callee r

```
void final(){
  int y,z;

  y = 2; z = 3;

  z = y+z;

}
```

2 caller r

Total: 6 STUR / 6 LDUR

LEGv8 ABI- Application Binary Interface

- The ABI is an agreement about how to use the various registers
- Not enforced by hardware, just a convention by programmers / compilers
- If you won't your code to work with other functions / libraries, **follow these**
- Some register conventions in ARMv8
  - X30 is the **link register** – used to hold return address
  - X28 is **stack pointer** – holds address of top of stack
  - X19-X27 are **callee-saved** – function must save these before writing to them
  - X0-15 are **caller-saved** –function must save live values before call
  - X0-X7 used for **arguments** (memory used if more space is needed)
  - X0 used for **return value**

Next Time

- Finish Up Function Calls
- Talks about linking – the final puzzle piece of software